XML Storage

Denilson Barbosa, Phil Bohannon, Juliana Freire, Carl-Christian Kanne, Ioana Manolescu, Vasilis Vassalos, Masatoshi Yoshikawa

X

XML Storage

Denilson Barbosa[1], Philip Bohannon[2], Juliana Freire[3], Carl-Christian Kanne[4], Ioana Manolescu[5], Vasilis Vassalos[6], Masatoshi Yoshikawa[7]
[1]Department of Computer Science, University of Calgary, Calgary, AB, Canada
[2]Yahoo! Research, CA, USA
[3]School of Computing, University of Utah, Salt Lake City, UT, USA
[4]Department of Computer Science, University of Manheim, Manheim, PA, USA
[5]INRIA Futurs, Le Chesnay, France
[6]Department of Informatics, Athens University of Economics and Business, Athens, Greece
[7]University of Kyoto, Japan

Synonyms

XML persistence; XML database

Definition

A wide variety of technologies may be employed to physically persist XML documents for later retrieval or update, from relational database management systems to hierarchical systems to native file systems. Once the target technology is chosen, there is still a large number of *storage mapping* strategies that define how parts of the document or document collection will be represented in the back-end technology. Additionally, there are issues of optimization of the technology and strategy used for the mapping. XML Storage covers all the above aspects of persisting XML document collections.

Historical Background

Even though the need for XML storage naturally arose after the emergence of XML, similar techniques had been development earlier, since the mid-1990's, to store semi-structured data ([EDS reference: Semi-structured data]). For example, the LORE system included a storage manager specifically designed for semi-structured objects, while the STORED system allowed the definition of mappings from semi-structured data to relations. Even earlier, storage techniques and storage systems had been developed for object-oriented data ([EDS reference: Object data models]). These techniques focused on storing individual objects, including their private and public data and their methods. Important tasks included performing garbage collection, managing object migration and maintaining class extents. Object clustering techniques were developed that used the class hierarchy and the *composition* hierarchy (i.e., which object is a component of which other object) to help determine object location. These techniques, and the implemented object storage systems, such as the O2 storage system, influenced the development of subsequent semi-structured and XML storage systems.

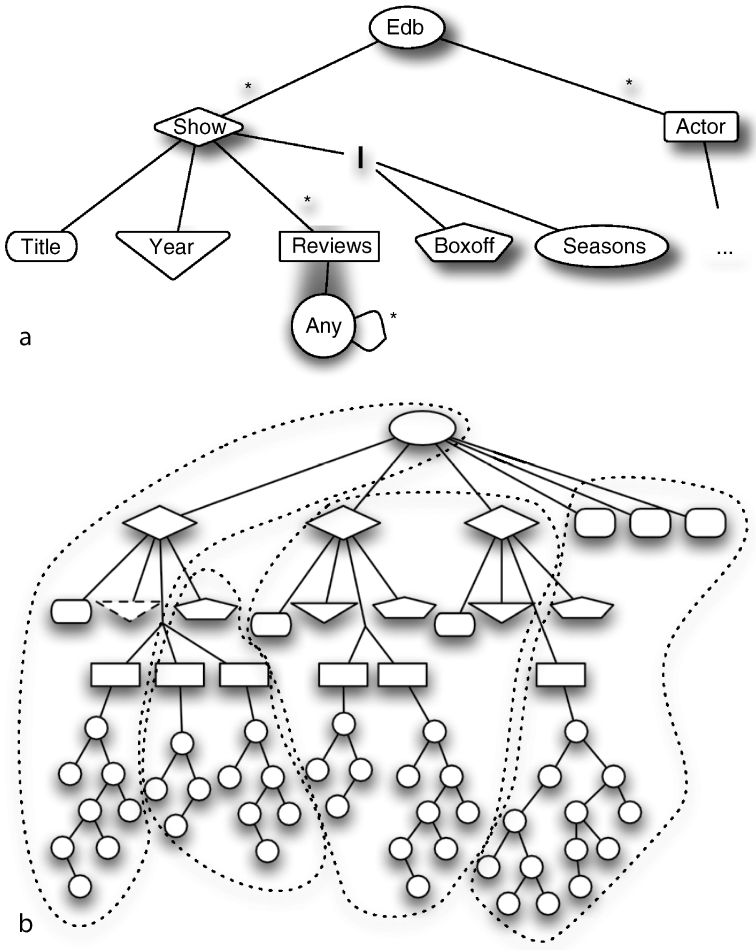
Moreover, the above solutions or ad-hoc approaches had also been used for the storage of large SGML (Standard Generalized Markup Language, a superset and precursor to XML) documents.

Scientific Fundamentals

Given the wide use of XML, most applications need or will need to process and manipulate XML documents, and many applications will need to store and retrieve data from large documents, large collections of documents, or both. As an exchange format, XML can be simply serialized and stored in a file, but serialized document storage often is very inefficient for query processing and updates. As a result, a large-scale XML storage infrastructure is critical to modern application performance.

Figure 1a shows a simple graphical outline of an XML DTD ([EDS reference]) for movies and television shows.

As this example shows, XML data may exhibit great variety in their structure. At one extreme, relational-style data like `title`, `year` and `boxoff` children of `show` in Fig. 1a may be represented in XML. At the opposite extreme are highly irregular structures such as might be found under the `reviews` tag. Figure 2 shows a similar graphical representation of a real-life DTD for



XML Storage. Figure 1. Movie DTD and example native storage strategy.

scientific articles. Since every HTML structure or formatting element is also an XML element or attribute, the corresponding XML tree is very deep and wide, and no two sections are likely to have the same structure.

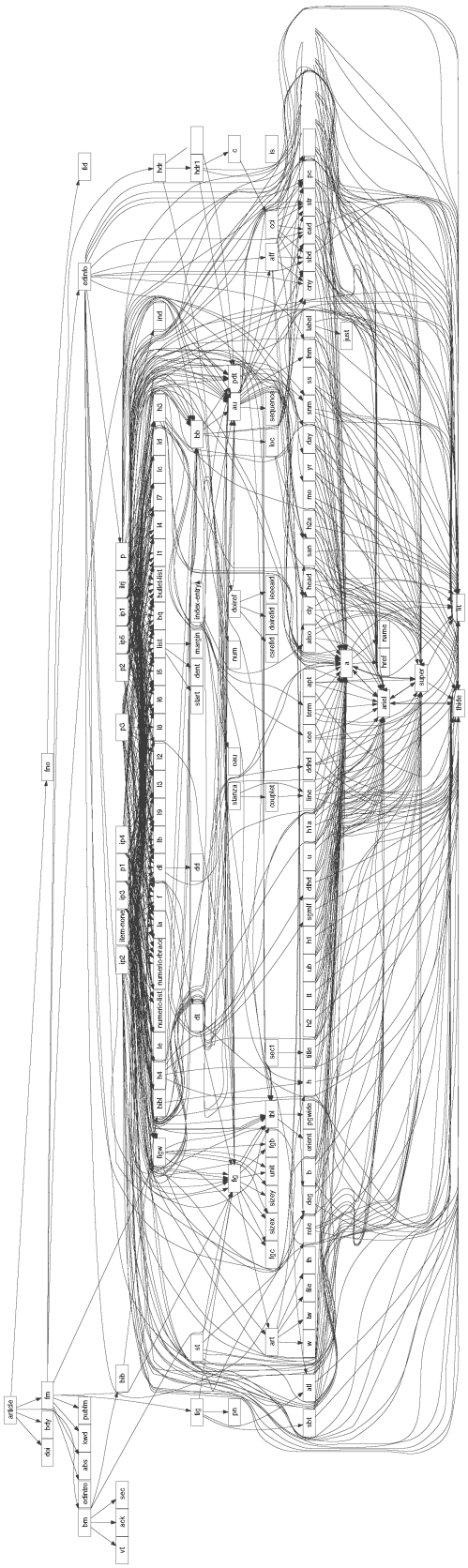
XML processing workloads are also diverse. Queries and updates may affect or return few or many nodes. They may also need to "visit" large portions of an XML tree and return nodes that are far apart, such as all the box-office receipts for movies, or may only return or affect nodes that are "close" together, such as all the information pertaining to a single review.

A few different ways of persisting XML document collections are used, and each addresses differently the challenges posed by the varied XML documents and workloads.

Instance-Driven Storage

In *instance-driven storage*, the storage of XML content is driven by the tree structure of the individual document, ignoring the types assigned to the nodes by a schema (if one exists). In some cases, e.g., when documents have irregular structure or an application mostly generates navigations to individual elements, instance-driven storage can greatly simplify the task of storing XML content. One instance-driven technique is to store nodes and edges in one or more relational tables. A second approach is to implement an XML data model natively.

Tabular Storage of Trees A relational schema for encoding any XML instance may include relations `child`, modeling the parent-child relationship, and `tag, attr, id, text`, associating to each element node respectively a tag, an attribute, an identity and a text value, as well as sets that contain the root of the document and the set of all its elements. Notice that such a schema does not allow full reconstruction of an



XML Storage. Figure 2. Graphical outline of a complex article DTD (strongly simplified).

original XML document, as it does not retain information on element order, whitespace, comments, entity references etc. The encoding of element order, which is a critical feature of XML, is discussed later in this article.

A relational schema for encoding XML may also need to capture *built-in integrity constraints* of XML documents, such as the fact that every child has a single parent, every element has exactly one tag, etc.

Tabular storage of trees as described enables the use of relational storage engines as the target storage technology for XML document collections. While capable of storing arbitrary documents, with this approach a large number of joins may be required to answer queries, especially when reconstructing subtrees. This is the basic storage mapping supported by Microsoft's SQL Server as of 2007.

Native XML Storage Native XML storage software implements data structures and algorithms specifically designed to store XML documents on secondary memory. These data structures support one or more of the XML data models ([EDS reference to infoset/psvi/XQuery data models]). Salient functional requirements implied by standard data models include the preservation of child order, a stable node identity, and support for type information (depending on the data model supported). An additional functional requirement in XML data stores is the ability to reconstruct the exact textual representation of an XML document, including details such as encoding, whitespace, attribute order, namespace prefixes, and entity references.

A native XML storage implementation generally maps tree nodes and edges to storage blocks in a manner that preserves *tree locality*, i.e., that stores parents and children in the same block. The strategy is to map XML tree structures onto records managed by a storage manager for variable-size records. One possible approach is to map the complete document to a single Binary Large Object and use the record manager's large object management to deal with documents larger than a page. This is one of the approaches for XML storage supported by the commercial DBMS Oracle as of 2007. This approach incurs significant costs both for update and for query processing.

A more sophisticated strategy is to divide the document into partitions smaller than a disk block and map each partition to a single record in the underlying store. Large text nodes and large child node lists are handled by chunking them and/or introducing auxiliary nodes. This organization supports efficient local navigation and tree reconstruction without, for example, loading the entire tree into memory. Such an approach is used in the commercial DBMS IBM DB2 as of 2007 (starting with version 9). Native stores can support efficiently updates, concurrency control mechanisms and traditional recovery schemes to preserve durability ([EDS reference ACID Properties]).

Figure 1b shows a hypothetical instance of the schema of Fig. 1a. The types of nodes are indicated by shape. One potential assignment of nodes to physical storage records is shown as groupings inside dashed lines. Note that `show` elements are often physically stored with their `review` children, and reviews are frequently stored with the next or previous review in document order.

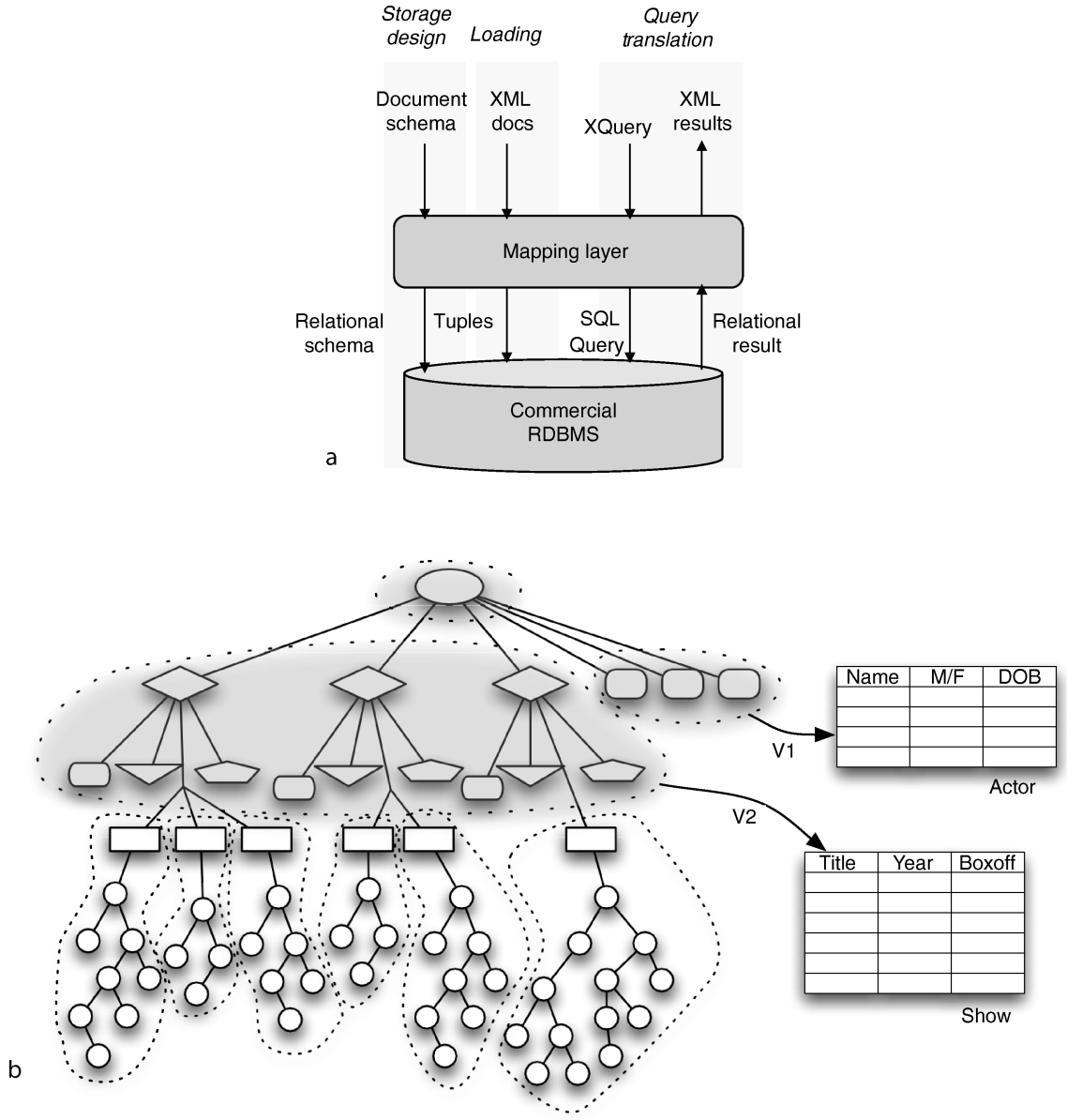
Physical-level heuristics that can be implemented to improve performance include compressed representation of node pointers inside a block, and string dictionaries allowing integers to replace strings appearing repeatedly, such as tag names and namespace URIs.

Schema-Driven Storage

When information about the structure of XML documents is given, e.g., in a DTD or an XML Schema ([EDS references to XML Schema]), techniques have been developed for XML storage that exploit this information. In general, nodes of the same type according to the schema are mapped in the same way, for example to a relational table. Schema information is primarily exploited for tabular storage of XML document collections, and in particular in conjunction with the use of a relational storage engine as the underlying technology, as described in the next paragraph. In *hybrid* XML storage different data models, and potentially even different systems, store different document parts.

Relational Storage for XML Documents

Techniques have been developed that enable the effective use of a relational database management system to store XML. Figure 3a illustrates the main tasks that must be performed for storing XML in relational databases. First, the schema of the XML document is mapped into a suitable relational schema that can preserve the information in the original XML documents (Storage Design). The resulting relational schema needs



XML Storage. Figure 3. Relational storage workflow and example.

to be optimized at the physical level, e.g., with the selection of appropriate file structures and the creation of indices, taking into account the distinctive characteristics of XML queries and updates in general and of the application workload in particular. XML documents are then shredded and loaded into the flat tables (Data Loading). At runtime, XML queries are translated into relational queries, e.g. in SQL, submitted to the underlying relational system and the results are translated back into XML (Query Translation). ([EDS references to XML publishing, XML query processing]) Schema-driven relational storage mappings for XML documents are supported by the Oracle DBMS.

An XML-to-relational mapping scheme consists of *view definitions* that express what data from the XML document should appear in each relational table and constraints over the relational schema. The views generally map elements with the same type or tag name to a table and define a *storage mapping*. For example, in Fig. 3b, two views, `V1` and `V2` are used to populate the `Actors` and `Shows` tables respectively. A particular set of storage views and constraints along with physical

storage and indexing options together comprise a *storage design*. The process of parsing an XML document and populating a set of relational views according to a storage design is referred to as *shredding*.

Due to the mismatch between the tree-structure of XML documents and the flat structure of relational tables, there are many possible storage designs. For example, in Fig. 3b, if an element, such as `show`, is guaranteed to have only a single child of a particular type, such as `seasons`, then the child type may optionally be *inlined*, i.e., stored in the same table as the parent.

On the other hand, due to the nature of XML queries and updates, certain indexing and file organization options have been shown to be generally useful. In particular, the use of B-tree indexes (as opposed to hash-based indexes) is usually beneficial, as the translation of XML queries into relational languages often involves range conditions. There is evidence that the best file organization for the relations resulting from XML shredding is index-organized tables ([EDS reference to Index Creation and File Structure]), with the index on the attribute(s) encoding the order of XML elements. With such file organization, index scanning allows the retrieval of the XML elements in document order, as required by XPath semantics, with a minimum number of random disk accesses. The use of a path index that stores complete root-to-node paths for all XML elements also provides benefits.

Cost-Based Approaches A key quality of a storage mapping is efficiency – whether queries and updates in the workload can be executed quickly. Cost-based mapping strategies can derive mappings that are more efficient than mappings generated using fixed strategies. In order to apply such strategies, statistics on the values and structure of an XML document collection need to be gathered. A set of transformations and annotations can be applied to the XML schema to derive different schemas that result in different relational storage mappings, for example by merging or splitting the types of different XML elements, and hence mapping them into the same or different relational tables. Then, an efficient mapping is selected by comparing the estimated cost of executing a given application workload on the relational schema produced by each mapping. The optimizer of the relational database used as storage engine can be used for the cost estimation. Due to the size of the search space for mappings generated by the schema transformations, efficient heuristics are needed to reduce the cost without missing the most efficient mappings. Physical database design options, such as vertical partitioning of relations and the creation of indices, can be considered in addition to logical database design options, to include potentially more efficient mappings in the search space.

The basic principles and techniques of cost-based approaches for XML storage are shared with relational cost-based schema design.

Correctness and Losslessness An important issue in designing mappings is correctness, notably, whether a given mapping preserves *enough* information. A mapping scheme is *lossless* if it allows the reconstruction of the original documents, and it is *validating* if all legal relational database instances correspond to a valid XML document. While losslessness is enough for applications involving only queries over the documents, if documents must conform to an XML schema and the application involves both queries and updates to the documents, schema mappings that are validating are necessary. Many of the mapping strategies proposed in the literature are (or can be extended to be) lossless. While none of them are validating, they can be extended with the addition of constraints to only allow updates that maintain the validity of the XML document. In particular, even though losslessness and validation are undecidable for a large class of mapping schemes, it is possible to guarantee information preservation by designing mapping procedures which guarantee these properties by construction.

Order Encoding Schemes Different techniques have been proposed to preserve the structure and order of XML elements that are mapped into a relational schema. In particular, different labeling schemes have been proposed to capture the positional information of each XML element via the assignment of node labels. An important goal of such schemes is to be able to express structural properties among nodes, e.g., the child, descendant, following sibling and other relationships, as conditions on the labels. Most schemes are either *prefix-based* or *range-based* and can be used with both schema-driven and instance-based relational storage of XML.

In prefix-based schemes, a node's label includes as a prefix the label of its parent. Dewey-based order encodings are the best known prefix-based schemes.

The Dewey Decimal Classification was originally developed for general knowledge classification. The basic *Dewey-based encoding* assigns to each node in an XML tree an identifier that records the position of a node among its siblings, prefixed by the identifier of its parent node. In Fig. 1b, the Dewey-based encoding would assign the identifier 1.1.2 to the dashed-line `year` element. In range-based order encodings, such as *interval* or *pre/post encoding*, a unique {*start*,*end*} interval identifies each node in the document tree. This interval can be generated in multiple ways. The most common method is to create a unique identifier, *start*, for each node in a preorder traversal of the document tree, and a unique identifier, *end*, in a postorder traversal. Additionally, in order to distinguish children from descendants, a level number needs to be recorded with each node.

An important consideration for any order-encoding scheme is to be able to handle updates in the XML documents, and many improvements have been made to the above basic encodings to reduce the overhead associated with updates.

Hybrid XML Storage

Some XML documents have both very structured and very unstructured parts. This has lead to the idea of *hybrid* XML storage, where different data models, and even systems using different storage technologies, store different document parts. For example, in Fig. 3b, `review` elements and their subtrees can be stored very differently from `show` elements, for example by serializing each review according to the dashed lines in the figure or storing them in a native XML storage system.

Prototype systems such as MARS and XAM have been proposed that support a hybrid storage model at the system level, i.e., provide physical data independence. In these systems, different access methods corresponding to the different storage mappings are formally described using views and constraints, and query processing involves the use of query rewriting using views ([EDS reference to Query rewriting using views]). Moreover, an appropriate tool or language is necessary to specify hybrid storage designs effectively and declaratively.

An additional consideration in favor of hybrid XML storage is that storing some information redundantly using different techniques can improve the performance of querying and data retrieval significantly by combining their benefits. For example, schema-directed relational storage mappings often give better performance for identifying the elements that satisfy an XPath query, while native storage allows the direct retrieval of large elements. In environments where updates are infrequent or update cost less important than query performance, such as various web-based query systems, such redundant storage approaches can be beneficial.

Key Applications

XML Storage techniques are used to efficiently store XML documents, XML messages, accumulated XML streams and any other form of XML-encoded content. XML Storage is a key component of an XML database management system. It can also provide significant benefits for the storage of semi-structured information with mostly tree structure, including scientific data.

Cross-references

- ► Dataguides
- ► Deweys
- ► Intervals
- ► Storage Management
- ► XML Document
- ► XML Indexing
- ► XML Query Processing
- ► XML Schema
- ► XPath/XQuery

Recommended Reading

1. Arion A., Benzaken V., Manolescu I., and Papakonstantinou Y. Structured materialized views for XML queries. In VLDB. Vienna, Austria, 2007, pp. 87–98.
2. Barbosa D., Freire J., and Mendelzon A.O. Designing information-preserving mapping schemes for XML. In VLDB. VLDB Endowment, Trondheim, Norway, 2005, pp. 109–120.
3. Beyer K., Cochrane R.J., Josifovski V., Kleewein J., Lapis G., Lohman G., Lyle B., Özcan F., Pirahesh H., Seemann N., Truong T., der Linden B.V., Vickery B., and Zhang C. System RX: one part relational, one part XML. In SIGMOD Conference. ACM, New York, NY, USA, 2005, pp. 347–358.
4. Chaudhuri S., Chen Z., Shim K., and Wu Y. Storing XML (with XSD) in SQL databases: interplay of logical and physical designs. IEEE Trans. Knowl. Data Eng., 17(12):1595–1609, 2005.
5. Deutsch A., Fernandez M., and Suciu D. Storing semi-structured data with STORED. In SIGMOD Conference. ACM, New York, NY, USA, 1999, pp. 431–442.
6. Fiebig T., Helmer S., Kanne C.C., Moerkotte G., Neumann J., Schiele R., and Westmann T. Anatomy of a native XML base management system. VLDB J., 11(4):292–314, 2003.

7. Georgiadis H. and Vassalos V. XPath on steroids: exploiting relational engines for XPath performance. In SIGMOD Conference. ACM, New York, NY, USA, 2007, pp. 317–328.
8. Härder T., Haustein M., Mathis C., and Wagner M. Node labeling schemes for dynamic XML documents reconsidered. Data Knowl. Eng., 60(1):126–149, 2007.
9. McHugh J., Abiteboul S., Goldman R., Quass D., and Widom J. Lore: a database management system for semistructured data. SIGMOD Rec., 26:54–66, 1997.
10. Shanmugasundaram J., Tufte K., He G., Zhang C., DeWitt D., and Naughton J. Relational databases for querying XML documents: limitations and opportunities. In VLDB. Morgan Kaufmann, San Francisco, CA, USA, 1999, pp. 302–314.
11. Vélez F., Bernard G., and Darnis V. The O2 object manager: an overview. In Building an Object-Oriented Database System, The Story of O2. Morgan Kaufmann, San Francisco, CA, USA, 1992, pp. 343–368.